A Framework for Generation and Management of Self-Adaptive Enterprise Applications

Hugo Arboleda, Andrés Paz, Miguel Jiménez, and Gabriel Tamura

I2T/DRISO Research Group, Icesi University, Cali, Colombia
{hfarboleda,afpaz,majimenez,gtamura}@icesi.edu.co

Abstract. Demand for self-adaptive enterprise applications has been on the rise over the last years. Such applications are expected to satisfy context-dependent quality requirements in varying execution conditions. Their dynamic nature constitutes challenges with respect to their architectural design and development, and the guarantee of the agreed quality scenarios at runtime. In this paper we present SHIFT, a framework that integrates (i) an architecture for managing self-adaptive enterprise applications, (ii) automated derivation of self-adaptive enterprise applications and their respective monitoring infrastructure, and (iii) decision support for the assisted recomposition of self-adaptive applications.

Keywords: Self-adaptive enterprise applications, software product lines, component configurations.

1 Introduction

Enterprise applications (EAs) are intended to satisfy the needs of entire organizations and usually involve persistent data, concurrent user access to the information and several user interfaces to handle the big amount of data requested. They live in dynamic execution contexts and are no longer isolated but instead interacting with other systems. Their dynamic nature implies that they are constantly under the influence of external stimuli (*i.e.* disturbances) from various sources inside or outside the system scope that may affect their behaviour or the levels at which they satisfy agreed quality. Regardless of the intrinsic uncertainty of disturbances and their possible sources, EAs still have to fulfill the customers' quality agreements. This has generated a growing interest regarding support infrastructure for autonomic adaptation of EAs, as well as flexible architectural designs conceived for allowing recomposition at runtime.

In this paper we present preliminary results regarding our SHIFT framework, which provides (i) an architecture for managing self-adaptive enterprise applications based on the adaptation feedback loop of the DYNAMICO reference model [16], (ii) support for automated derivation of self-adaptive enterprise applications considering possible functional, quality and monitoring variations, and (iii) automated reasoning at runtime regarding context- and system-sensed data to determine and apply necessary system adaptations, considering deployment

and undeployment tasks. SHIFT's constituent elements are at different stages of development; throughout this paper we specify their advances.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation. Section 3 provides a general description of the SHIFT framework. Section 4 describes our concrete implementation of the architecture for managing self-adaptive enterprise applications. Section 5 presents the mechanisms for assisted derivation of applications. Section 6 presents our adaptation planning strategy based on automated reasoning. Section 7 sets out conclusions and outlines future work.

2 Background and Motivation

Current approaches implement dynamic adaptation of service compositions at the language level [7, 5, 13], or using models at runtime [6, 11, 12]. The first ones can be complex and time-consuming, and with low-level implementation mechanisms. Our work is related to approaches that use models at runtime. Model-based approaches for dynamic adaptation implement, tacit or explicitly, the MAPE-K reference model [9] that comprises five components: (i) a Monitor, (ii) an Analyzer, (iii) a Planner, (iv) an Executor, and (v) a Knowledge base. The recent work of Alferez *et al.* [1] summarizes good practices implementing the MAPE-K reference model and gives implementation details about reconfiguration mechanisms. They center their attention on service recomposition at runtime using (dynamic) product line engineering practices for assembling and redeploying complete applications according to context- and system-sensed data. Model-based approaches for dynamic adaptation of service compositions do not consider changing requirements over SCA composites or EJB models. This triggers new challenges given the complexity of deployment at the stage of adapting composites and EJB bindings. Such approaches also lack support for assisted derivation of monitoring infrastructures, which is important in order to manage reference architectures that prevent the infrastructure from introducing considerable overhead in the system's regular operations. Assisted derivation of monitoring infrastructure also guarantees relevance of the complete self-adaptive architecture in changing context conditions of system execution [16].

In previous work, we proposed independent approaches and implementations in the contexts of the engineering of highly dynamic adaptive software systems with the DYNAMICO reference model [16], quality of service (QoS) contract preservation under changing execution conditions with the QoS-CARE implementation [15], model-based product line engineering with the FieSta approach [4, 2, 8], automated reasoning for derivation of product lines [3], and the recent (unpublished) contributions regarding quality variations in the automated derivation process of product lines [8]. The SHIFT framework is motivated by the required integration of all these efforts in a move to approach automation and quality awareness along the life cycle of enterprise applications. With SHIFT we are currently focused in the design, development, deployment and operation

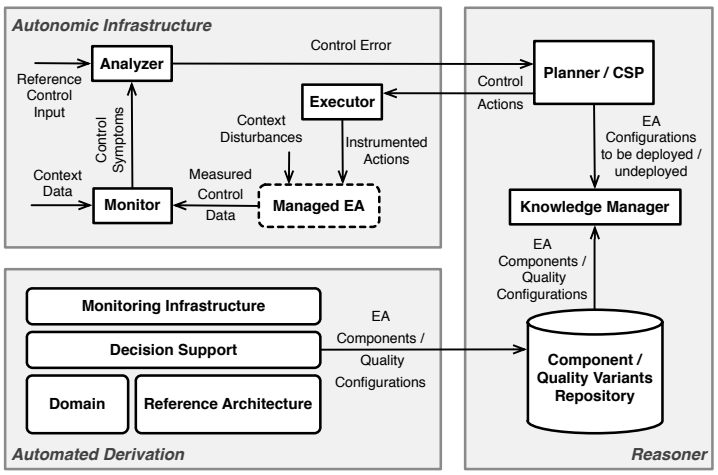

Fig. 1. High-level architectural view of the SHIFT components.

stages of the life cycle. The remaining stages (*e.g.*, testing, maintenance/evolution) are part of our ongoing research work.

3 Framework Overview

Figure 1 presents a high-level architectural view of the SHIFT constituting elements. The `Automated Derivation` element is concerned about providing support for functional and quality configuration and derivation of (i) deployable enterprise applications components and (ii) monitoring infrastructure. Generated components are stored in the `Component Repository`, which is managed by the `Reasoner` element; they are an input for (re)deployment processes. The monitoring infrastructure is deployed as part of the `Autonomic Infrastructure` element, which implements the adaptation feedback loop of the DYNAMICO reference model [16]. As part of the `Reasoner` element, SHIFT considers the need for dynamically deploying and undeploying components to realize adaptation plans. In order to obtain the best possible selection of components when configuring a deployable product, we rely on Constrain Programming to reason on the set of constraints defined by reachable configurations and their relationships with the component repository. Our deployment considers transporting components from their source repository to the corresponding computational resource, undeploying previous versions of them, deploying them into the middleware or application server, binding their dependencies and services, and executing them. In addition, if necessary, to recompile system source code to make measurement interfaces available to the monitoring infrastructure. Accordingly, these deployment tasks are applied to the monitoring components to effectively ensure dynamic quality awareness.

4 Autonomic Infrastructure

The autonomic manager, based on the MAPE-K reference model, is the infrastructure that allows the derived EAs to be adapted to unforeseen context changes in order to ensure the satisfaction of agreed Service Level Agreements (SLA). Composing this infrastructure is a `Monitor` component that continuously senses relevant context data, an `Analyzer` that interprets monitoring events reported by the `Monitor` to determine whether the SLAs are being fulfilled, and the `Planner` and `Executor` components that synthesise and realize adaptation plans to alter the system's behavior, either by modifying the system structure or by varying parameters to reach a desired system state [9]. These four components share relevant information through the `Knowledge` source component. Our current implementation considers the automated derivation of the monitoring infrastructure that realizes monitoring components, comprising *in situ* monitor probes and event-based monitors that collect context data. The analyzer is subscribed to attend monitoring events from monitors, and is in charge of deciding when an adaptation is needed to ensure the fulfillment of the performance SLAs. Our planner and knowledge implementations (*cf.* `Reasoner` in Figure 1) follow a constraint programming approach to find the best configuration of components necessary to preserve the satisfaction of the performance SLAs, when available. Finally, the `Executor` component realizes the adaptation plan produced by the planner, redeploying SCA and EJB components by means of the introspection capabilities in the FraSCAti middleware [14] and the *application versioning* feature in the GlassFish application server, respectively.

5 Automated Derivation of Applications and Monitors

5.1 Specification and Design of Functionality and Quality

Functional Variations. The `Automated Derivation` element in Figure 1 is divided into four subelements: the `Domain`, the `Reference Architecture`, the `Decision Support` and the `Monitoring Infrastructure`. The domain subelement comprises an extensible metamodel for capturing the functional scope of product lines in the context of enterprise applications. This is based on our previous work [8]. The metamodel captures the variability in terms of business entities and their relationships, enabling the management of functional variability that involves CRUD operations over the entities, considering one-to-many and one-to-one relationships between them.

Quality Variations. Quality variations are modeled as scenarios in the form of variation points (*i.e.* stimulus) and variants (*i.e.* alternative responses to stimulus) by using support tool from the `Decision Support` subelement. The `Reference Architecture` subelement is focused on supporting the modeling of architectural implementations for quality variations. In order to associate architectural implementations for quality variants, we select design patterns in their pure form or we compose them. Resulting structures are documented as variable reference architecture fragments that are later composed and become concrete

during the derivation process of components and complete applications. In that way, we compose patterns respecting a base (common) reference architecture, over which variable reference architecture fragments are integrated before deriving concrete implementations. By exploiting the relationships between the `Domain` and `Decision Support` subelements, product line engineers may accurately model the impacts of functional variants on quality attributes, and vice versa, when they exist. In order to manage the suitable relationships between functional and quality variations, from our previous work [4, 2] we consider the need of constraining the bindings between both submodels. The strategy is based on defining OCL restrictions for capturing and validating constraint logic.

Designing Concrete Architectures. The `Decision Support` subelement provides support for assisted reasoning regarding achievable configurations and their interactions. Our decision model is a collection of (partial) reachable product configurations, expressed as sets of quality variants, and the modeling of their impact on other configurations. The impact of one configuration over another is expressed in terms of *promote*, *require*, *inhibit* and *exclude* relationships. For every pair of related configurations, a reference architecture fragment should be associated. Such fragments model the resulting structures and behavior that produces the composition of patterns associated to variants involved in the related configurations. Concrete architectures of reusable components and complete applications are created as a composition of a common reference architecture and reference architecture fragments. Composition rules are managed in model-based artifacts that will be introduced in the following section.

5.2 Component Derivation

Reusable components and complete applications result from transforming a set of functionalities contained in a domain model along with a configuration of quality levels into source code. The transformation process satisfies the constraints and conditions dictated by a common reference architecture and the variable reference architecture fragments that contribute. Our generation strategy is based on the delegation of responsibilities for composing templates (*i.e.* model2text transformations) in order to weave common and variable abstract reference architecture fragments. The common reference architecture is associated to a set of controller templates that are in charge of orchestrating the concrete architecture composition. Such controllers know the specific point where a contribution is needed, plus the concrete contribution it requires according to possible variants. Thus, controllers delegate the code declaration to contributors, which are concrete Java classes able to return final source code or delegate on other contributors the responsibility of returning required source code. The generation process also creates templates and contributors for quality variations and related reference architecture fragments. We developed a common library as tool support for describing and weaving required compositions. The library includes a language for describing the required composition, and also includes an engine for weaving code fragments. Currently, we generate JEE7 components under

the EJB 3.2 specification. The generation of SCA composites is under development. The specification, design and derivation of quality-concerned enterprise application is part of our recent (unpublished) work available in [8].

5.3 Specification and Derivation of Monitoring Infrastructure

In SHIFT, the specification and generation of monitoring components, deployable at runtime, is performed through PaSCAni. PaSCAni is a Domain Specific Language (DSL) that allows defining two types of components: monitor probes (implemented either as an EJB or SCA component) and monitors (implemented as a SCA component). The first are introduced into the system, binding them appropriately and acting as a sensor, therefore allowing to measure actual service executions. Thus, the DSL allows system administrators to monitor components that were not considered to be monitored before the initial system deployment. The second contain the necessary logic to abstract single context events (*i.e.* events arising from monitor probes) into complex and relevant monitoring data to be analyzed by the Analyzer and other components (*e.g.*, log components and monitoring dashboards). Both monitor probes and monitors are supplied with standard traceability and controllability mechanisms to (i) prevent the monitoring infrastructure from introducing considerable overhead in the system's regular operations, and (ii) feed knowledge sources with relevant monitoring data. Controlling the produced monitoring components is important when the system reaches critical quality levels, given that it can end up breaching quality agreements or overusing system resources.

The interaction between probes and monitors is event-based, and is specified in a single source file. Monitoring specifications can be parametrized and derived in an automated way for any system component, for those quality attributes with clear definition and already proposed metrics and measurement methods [10]. In SHIFT's current implementation, we have already designed a mechanism for automatically generating PaSCAni specifications for the performance quality attribute. This mechanism takes place in the automated derivation phase, and produces the monitoring specification and its corresponding deployment descriptors.

In order to monitor EAs, we consider EJB components in our DSL specification, in a way that monitor probes can be integrated with EJB implementations following Aspect Oriented Programming. EJB probes communicate with SCA monitors through Web Service bindings, accomplishing the same functionality that SCA probes. Service interception is realized by means of `Intent` composites, in FraSCAti, and `Interceptors` in the GlassFish application server. Regarding the adaptation of the monitoring infrastructure at runtime, the dynamic deployment is realized by using the dynamic reconfiguration API in FraSCAti, and the *application versioning* feature in GlassFish.

6 Automated Reasoning for Components Deployment

The `Reasoner` element in Figure 1 includes the `Planner/CSP` subelement, which is in charge of designing the adaptation plans to alter the system's behavior by modifying its structure or by varying parameters to reach a desired system state. In order to obtain the best possible selection of components to alter the system's behavior, we use Constrain Programming (CP) to reason on the set of constraints defined by reachable configurations and their relationships with the component repository. A configuration is a finite set of variants with an unselected or selected state. We relate on decision models the information of components and variants in order to define the necessary actions to derive deployable components in accordance with a configuration. Implementing a variant in an application may often require several components, thus, we refer as a *componentset* to the set of components implementing a variant. A decision model is a finite set of decisions. Each decision relates one *componentset* with one variant. A resolution model is a decision model instance, which binds variability and defines how to derive one product. A resolution model is a finite set of *componentset* applications. The application is not planned if the *componentset* should not be deployed, and planned if the *componentset* should be deployed. However, not every possible resolution model is a valid resolution model. A valid resolution model must satisfy the following constraints: A *componentset* must be deployed satisfying the respective planned application in the decision model. Two deployable *componentset*s must not exclude each other. All applicable *componentset*s must take into account all the variants' states in the configuration.

Our current implementation of the `Planner/CSP` subelement, allow us to answer the following questions: *application*, *possible resolutions*, *number of resolutions*, *validation*, *flexible componentsets*, *inflexible componentsets*, and *optimum resolution*.

7 Conclusions and Future Work

In this paper we have presented our advances on SHIFT, a framework for generation and management of self-adaptive enterprise applications. We discussed our architecture, which is based on the MAPE-K and DYNAMICO reference models [9, 16]. SHIFT includes three elements: `Autonomic Infrastructure`, `Automated Derivation`, and `Reasoner`. We discussed how SHIFT offers automated reasoning as support for adequate (re)deployment executions by using a `Planner/CSP` subelement that is implemented by using Constraint Programming. We have illustrated how SHIFT considers changing requirements over SCA composites and EJB models, and offer support for assisted derivation of enterprise applications and monitoring infrastructures. As future work, we keep working on the concrete implementations of the SHIFT elements, including the complete autonomic infrastructure and its interoperability with JEE middlewares.

Acknowledgments

This work has been partially supported by grant 0369-2013 from the Colombian Administrative Department of Science, Technology and Innovation (Colciencias) under project SHIFT 2117-569-33721.

References

1. G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz. Dynamic adaptation of service compositions with variability models. *Systems and Software*, 91(1):24–47, 2014.
2. H. Arboleda, R. Casallas, and J.-C. Royer. Dealing with Fine-Grained Configurations in Model-Driven SPLs. In *Proc. of the SPLC'09*, pages 1–10, San Francisco, CA, USA, Aug. 2009. Carnegie Mellon University.
3. H. Arboleda, J. F. Díaz, V. Vargas, and J. Royer. Automated reasoning for derivation of model-driven spls. In *SPLC'10 MAPLE'10*, pages 181–188, 2010.
4. H. Arboleda and J.-C. Royer. *Model-Driven and Software Product Line Engineering*. ISTE-Wiley, 1st edition, 2012.
5. L. Baresi and S. Guinea. Self-supervising bpel processes. *IEEE Trans. on Software Engineering*, 37(2):247–263, 2011.
6. R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Trans. on Software Engineering*, 37(3):387–409, 2011.
7. M. Colombo, E. Di Nitto, and M. Mauri. Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In *Proc. of the ICSOC'06*, pages 191–202. Springer, 2006.
8. D. Durán and H. Arboleda. Quality-driven software product lines. Master's thesis, Icesi University, 2014.
9. IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, 2006.
10. ISO/IEC. ISO/IEC 25000 - Guide to SQuaRE. Technical report, 2014.
11. D. Menasce, H. Gomaa, S. Malek, and J. P. Sousa. Sassy: A framework for self-architecting service-oriented systems. *IEEE Software*, 28(6):78–85, 2011.
12. B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Model driven engineering languages and systems*, pages 782–796. Springer, 2008.
13. N. C. Narendra, K. Ponnalagu, J. Krishnamurthy, and R. Ramkumar. *Run-time adaptation of non-functional properties of composite web services using aspect-oriented programming*. Springer, 2007.
14. L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience (SPE)*, pages 1–26, 2012.
15. G. Tamura, R. Casallas, A. Cleve, and L. Duchien. QoS contract preservation through dynamic reconfiguration: A formal semantics approach. *Science of Computer Programming*, 94:307–332, 2014.
16. N. M. Villegas, G. Tamura, H. A. Müller, L. Duchien, and R. Casallas. DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems. *LNCS*, 7475:265–293, 2013.